VAXclusters: A Closely-Coupled Distributed System

NANCY P. KRONENBERG, HENRY M. LEVY, and WILLIAM D. STRECKER
Digital Equipment Corporation

A VAXcluster is a highly available and extensible configuration of VAX computers that operate as a single system. To achieve performance in a multicomputer environment, a new communications architecture, communications hardware, and distributed software were jointly designed. The software is a distributed version of the VAX/VMS operating system that uses a distributed lock manager to synchronize access to shared resources. The communications hardware includes a 70 megabit per second message-oriented interconnect and an interconnect port that performs communications tasks traditionally handled by software. Performance measurements show this structure to be highly efficient, for example, capable of sending and receiving 3000 messages per second on a VAX-11/780.

Categories and Subject Descriptors: C.2.5 [**Computer Communications Networks**]: Local Networks—*buses*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.4 [**Operating Systems**]: Communications Management; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; D.4.7 [**Operating Input/Output Systems**]: Organization and Design—*performance*

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Device control protocols, distributed database management, intersystem communication protocols, network protocols, resource locking

1. INTRODUCTION

Contemporary multicomputer systems typically lie at the ends of the spectrum delimited by tightly-coupled multiprocessors and loosely-coupled distributed systems. Loosely-coupled systems are characterized by physical separation of processors, low-bandwidth message-oriented interprocessor communication, and independent operating systems [1, 2, 5, 13]. Tightly-coupled systems are characterized by close physical proximity of processors, high-bandwidth communication through shared memory, and a single copy of the operating system [7, 9, 15].

An intermediate approach taken at Digital Equipment Corporation was to build a "closely-coupled" structure of standard VAX computers [16], called *VAXclusters*. By closely-coupled, we imply that a VAXcluster has characteristics of both loosely- and tightly-coupled systems. On one hand, a VAXcluster has separate processors and memories connected by a message-oriented interconnect, running separate copies of a distributed VAX/VMS operating system. On the

Authors' addresses: N. P. Kronenberg and W. D. Strecker, Digital Equipment Corporation, 295 Foster Street, Littleton, MA 01460; H. M. Levy, Department of Computer Science, University of Washington, Seattle, WA 98195.

other hand, the cluster relies on close physical proximity, a single (physical and logical) security domain, shared physical access to disk storage, and high-speed memory-to-memory block transfers between nodes.

The goals of the VAXcluster multicomputer system are high availability and easy extensibility to a large number of processors and device controllers. In contrast to other highly available systems [3, 4, 10, 11], VAXcluster is built from general-purpose, off-the-shelf processors and a general-purpose operating system. A key concern in this approach is system performance. Two important factors in the performance of a multicomputer system are the software overhead of the communications architecture and the bandwidth of the computer interconnect. To address these issues, several developments were undertaken, including:

(1) A simple, low-overhead communications architecture whose functions are tailored to the needs of highly available, extensible systems. This architecture is called SCA (System Communication Architecture).
(2) A very high speed message-oriented computer interconnect, termed the CI (Computer Interconnect).
(3) An intelligent hardware interface to the CI, called the CI Port, that implements part of SCA in hardware.
(4) An intelligent, message-oriented mass storage controller that uses both the CI and the CI Port interface.

This paper describes the new communications hardware developed for VAXclusters, the hardware-software interface, and the structure of the distributed VAX/VMS operating system. The developments described in this paper are part of Digital's VAXcluster product; there are, as of early 1985, approximately 2000 VAXcluster systems in operation.

2. VAXcluster HARDWARE STRUCTURE

Figure 1 shows the topology of an example VAXcluster. The components of a VAXcluster include the CI, VAX hosts, CI Ports, and HSC-50 mass storage (i.e., disk and tape) controllers. For high reliability applications, a cluster must contain a minimum of two VAX processors and two mass storage controllers with dual-ported devices. The preferred method of attaching terminals is through a Local Area Terminal server (not shown in the figure), which allows a terminal to connect to any host in a VAXcluster.

The CI is a dual path serial interconnect with each path supporting a 70 Mbit/s transfer rate. The primary purpose of the dual paths is to provide redundancy in the case of path failure, but when both paths are available they are usable concurrently. Each path is implemented in two coaxial cables: one for transmitted and one for received signals. Baseband signaling with Manchester encoding is employed.

While the CI is logically a bus, it is physically organized as a star topology. A central hub called the Star Coupler connects all of the nodes through radial CI paths of up to 45 meters. The current coupler is a passive device that supports a maximum of 16 nodes; node addresses are 8 bits providing an architectural limit of 256 nodes.

The selection of a star topology was chosen over a conventional linear topology for several reasons. First, the efficiency of a serial bus is related to the longest

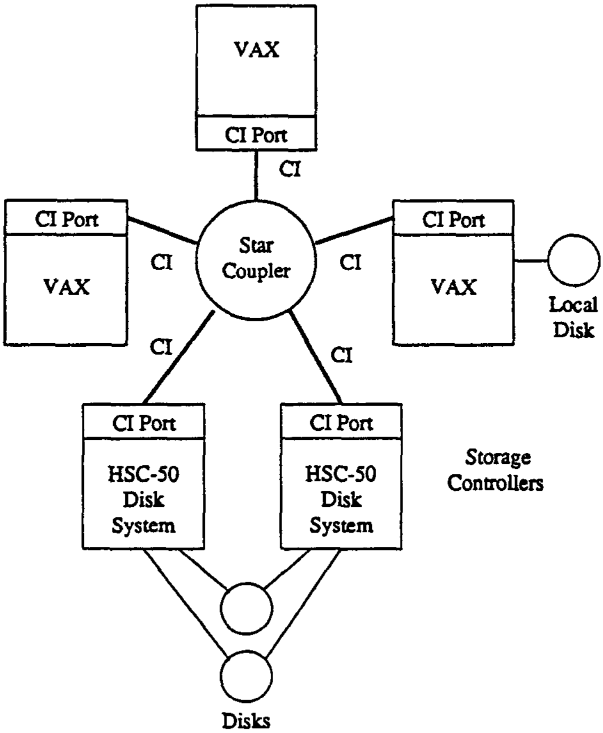


Fig. 1. VAXcluster hardware topology.

transit time between nodes. The star permits nodes to be located within a 45 meter radius (about 6400 square meters) with a maximum node separation of 90 meters. Typically, a linear bus threaded through 16 nodes in the same area would greatly exceed 90 meters. Second, the central coupler provides simple, electrically and mechanically safe addition and removal of nodes.

The CI port is responsible for arbitration, path selection, and data transmission. Arbitration uses carrier sense multiple access (CSMA) but is different from the arbitration used by Ethernet [12]. Each CI port has a node-specific delay time. When wishing to transmit, a port waits until the CI is quiet and then waits its specific delay time. If the CI is still quiet the node has won its arbitration and may send its packet. This scheme gives priority to nodes with short delay times. To ensure fairness, nodes actually have two delay times—one relatively short and one relatively long. Under heavy loading, nodes alternate between short and long delays. Thus, under light loading the bus is contention driven and under heavy loading it is round robin.

When a port wins an arbitration, it sends a data packet and waits for receipt of an acknowledgement. If the data packet is correctly received, the receiving port immediately returns an acknowledgement packet *without* rearbitrating the CI. This is possible because the CI port can generate an acknowledgement in less time than the smallest node-specific delay. Retries are performed if the sending CI port does not receive an acknowledgement.

To distribute transmissions across both paths of the dual-path CI, the CI port maintains a *path status table* indicating which paths to each node are currently good or bad. Assuming that both paths are marked good, the CI port chooses one randomly. This provides statistical load sharing and early detection of failures. Should repeated retries fail on one path, that path is marked bad in the status table and the other path is tried.

3. THE CI PORT ARCHITECTURE

Each cluster host or mass storage controller connects to the CI through a CI port. CI ports are device specific and have been implemented for the HSC-50 mass storage controller and the VAX 11/750, VAX 11/780, VAX 11/782, VAX 11/785, and VAX/8600 hosts. All CI ports implement a common architecture, whose goals are to:

(1) Offload much of the communications overhead typically performed by nodes in distributed systems.
(2) Provide a standard, message-oriented software interface for both interprocessor communication and device control.

The design of the CI port is based on the needs of the VMS System Communications Architecture. SCA is a software layer that provides efficient communications services to low-level distributed applications (e.g., device drivers, file services, and network managers). SCA supports three communications services: datagrams, messages, and block data transfers.

SCA datagrams and messages are information units of less than 4 kbytes sent over a connection. They differ only in reliability. Delivery of datagrams is not guaranteed; they can be lost, duplicated, or delivered out of order. Delivery of messages is guaranteed, as is their order of arrival. Datagrams are used for status and information messages whose loss is not critical, and by applications such as DECNET that have their own high-level reliability protocols. Messages are used, for example, to carry disk read and write requests.

To simplify buffer allocation, hosts must agree on the maximum size of messages and datagrams that they will transmit. VAXcluster hosts use standard sizes of 576 bytes for datagrams and 112 bytes for messages. These sizes are keyed to the needs of DECNET and the lock management protocol, respectively.

To ensure delivery of messages without duplication or loss, each CI port maintains a virtual circuit with every other cluster port. A virtual circuit descriptor table in each port indicates the status of its port-to-port virtual circuits. Included in each virtual circuit descriptor are sending and receiving sequence numbers. Each transmitted message carries a sequence number enabling duplicate packets to be discarded.

Block data is any contiguous data in a process' virtual address space. There is no size limit except that imposed by the virtual and physical memory constraints of the host. CI ports are capable of copying block data directly from process virtual memory on one node to process virtual memory on another node.

Delivery of block data is guaranteed. The sending and receiving ports cooperate in breaking up the transfer into data packets and ensuring that all packets are correctly transmitted, received, and placed in the appropriate destination buffer. Virtual circuit sequence numbers are used on the individual packets, as with messages. Thus, the major differences between block data and messages are the size of the transfer and the fact that block data does not need to be copied by the host operating system. Block data transfers are used, for example, by disk subsystems to move data associated with disk read and write requests.

3.1 CI Port Interface

The VAX CI Port interface is shown in Figure 2. The interface consists of a set of seven queues: four command queues, a response queue, a datagram free queue, and a message free queue. The queues and queue headers are located in host memory. When the port is initialized, host software loads a port register with the address of a descriptor for the queue headers.

Host software and the port communicate through queued command and response packets. To issue a port command, port driver software queues a command packet to one of the four command queues. The four queues allow for four priority levels; servicing is FIFO within each queue. An opcode within the packet specifies the command to be executed. The response queue is used by the port to enqueue incoming messages and datagrams, while the free queues are a source of empty packets for incoming messages and a sink for transmitted message packets.

For example, to send a datagram, software queues a SEND DATAGRAM packet onto one of the command queues. The packet contains an opcode field specifying SEND DATAGRAM, a port field containing the destination port number, the datagram size, and the text of the datagram. The packet is doubly linked through its first two fields. This structure is shown in Figure 3.

If host software needs confirmation when the packet is sent, it sets a *response queue* bit in the packet. This bit causes the port to place the packet in the response queue after the packet has been transmitted. The response packet is identical to the SEND DATAGRAM packet, except that the status field indicates whether the send was successful. Had the response queue flag bit been clear in the SEND DATAGRAM command (as it typically is), the port would instead place the transmitted command packet on the datagram free queue.

When a CI port receives a datagram, it takes a packet from its datagram free queue. Should the queue be empty, the datagram is discarded. Otherwise, the port constructs a DATAGRAM RECEIVED packet that contains the datagram text and the port number of the sending port. This packet is then queued on the response queue.

Messages operate in a similar fashion, except that they have a different opcode and message buffers are dequeued from the message free queue. If the message free queue is empty when a message arrives, the port generates an error interrupt to the host. High level SCA flow control ensures that the message free queue does not become empty.

Block transfer operations are somewhat more complicated. Each port has a data structure called a *buffer descriptor table.* Before performing a block transfer, host software creates a buffer descriptor that defines the virtual memory buffer to be used. The descriptor contains a pointer to the first VAX page table entry mapping the virtually contiguous buffer. In addition, the descriptor contains the offset (within the first page) of the first byte of the buffer, the length of the buffer, and a 16-bit key. The data structures for a block transfer are illustrated in Figure 4.

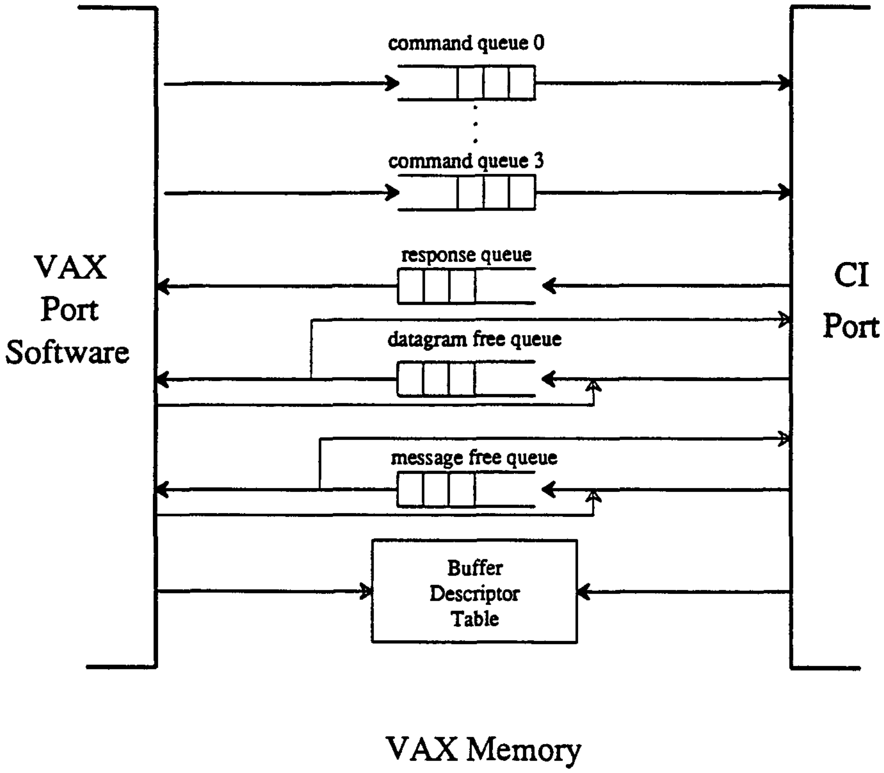
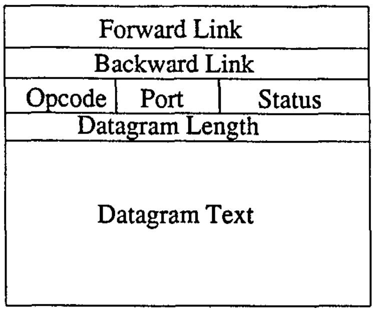


Fig. 2. CI Port interface.



Forward Link		
Backward Link		
Opcode	Port	Status
Datagram Length		
Datagram Text		

Fig. 3. Example CI Port command packet.

Each buffer has a 32-bit *name*. The name consists of a 16-bit buffer descriptor table index and the 16-bit buffer key. The key is used to prevent dangling references and is modified whenever a descriptor is released. To transfer block data, the initiating software must have the buffer names of the source and destination buffers. The buffer names are exchanged through a high level message protocol. A host can cause data to be moved either to another node (SEND DATA) or from another node (REQUEST DATA). A SEND DATA or REQUEST DATA command packet contains the names of both buffers and the length of the transfer. In

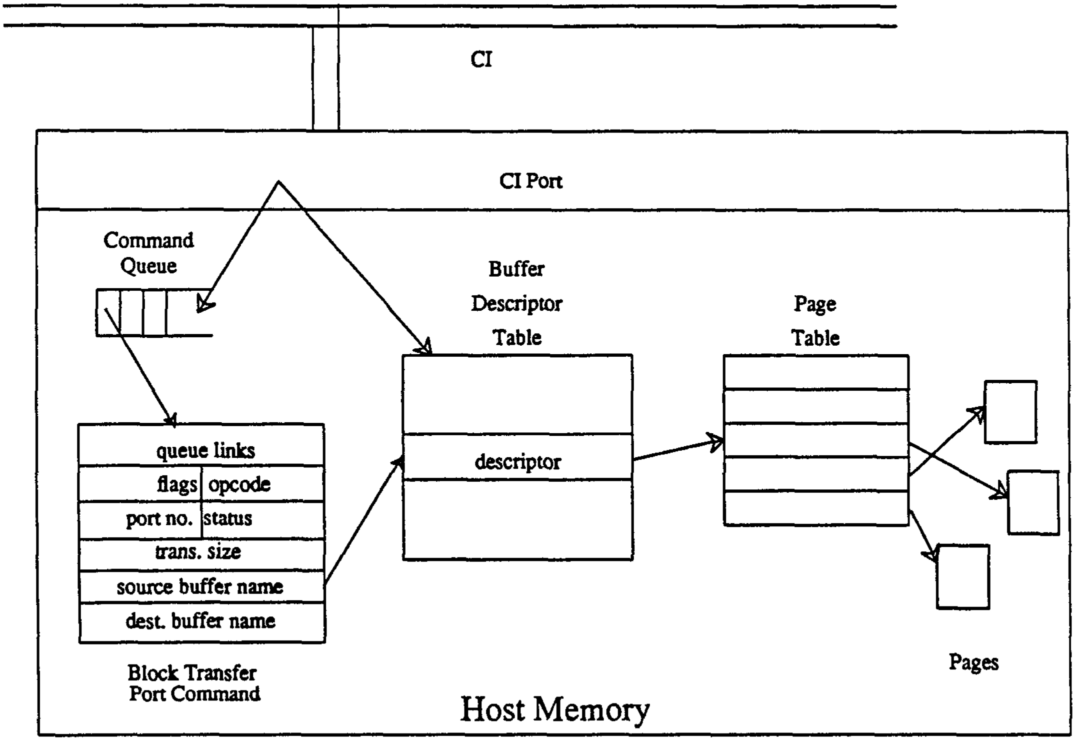


Fig. 4. CI Port block data memory mapping.

either case (send or request), a single command packet causes the source and destination ports to perform the block transfer. When the last packet has been successfully received, the initiating port places a response packet on its response queue indicating that the transfer is complete.

The goal of reducing VAX host interrupts is met through several strategies and mechanisms. First, the block transfer mechanism minimizes the number of interrupts necessary to transfer large amounts of data. Second, at the sending port, DATAGRAM SENT/MESSAGE SENT confirmation packets are typically generated only when a failure occurs. Third, a receiving port interrupts the VAX only when it queues a received packet on an empty response queue. Thus, when software dequeues a packet in response to an interrupt, it always checks for more packets before dismissing the interrupt.

4. MASS STORAGE CONTROL

The move from Control and Status Register activated storage devices to message-oriented storage devices offers several advantages, including:

(1) Sharing is simplified. Several hosts can queue messages to a single controller. In addition, device control messages can be transmitted to and executed by hosts with local disks.

(2) Ease of extension to new devices. In contrast to conventional systems, where there is a different driver for every type of disk and disk interface, a single *disk class driver* simply builds message packets and transmits them using a communications interface. The disk class driver is independent of drive specifics (e.g., cylinders and sectors). New disk and tape devices and controllers can be added with little or no modification to host software.

(3) Improved performance. The controller can maintain a queue of requests from multiple hosts and can optimize disk performance in real time. The controller can also handle error recovery and bad block replacement.

The HSC-50 (Hierarchical Storage Controller), shown in Figure 1, is the first CI-based controller for both disks and tapes. A single HSC controller can handle up to 24 disk drives. Multiple HSCs with dual-ported disks provide redundancy in case of failures.

The protocol interpreted by the HSC is called the Mass Storage Control Protocol (MSCP). The MSCP model separates the flow of control and status information from the flow of data. This distinction has been used in other systems to achieve efficient file access [6] and corresponds to the CI port's message and block data mechanisms; messages are used for device control commands while block transfers are used for data.

For example, to perform a disk read, the disk class driver transmits an MSCP READ message to the controller using a CI port SEND MESSAGE command. The read request contains the device type and unit number, the device media address (e.g., the disk logical block number), the 32-bit buffer name of the requester's buffer, and the length of the transfer. To process the request, the controller reads the specified data from disk and transmits it directly to host memory using a CI port SEND DATA command. When the data is successfully transferred, the controller sends a message to the host driver indicating the completion status of the request.

The same control protocol is used to provide cluster-wide access to CI-based controllers such as the HSC, and to Unibus or Massbus disks connected privately to a VAX node. Messages are routed from the disk class driver in the requesting node to an MSCP server on the node with the private disk. This server then parses the MSCP message, issues requests to its local disk, and initiates the block transfer through its SCA interface.

5. VAXcluster SOFTWARE

From a user's point of view, a VAXcluster is a set of nodes cooperating through VAX/VMS distributed operating system software to provide sharing of resources among users on all nodes. Shared resources include devices, files, records, and batch and print services. Typically, user account and password information resides in a single file shared by all cluster nodes. A user obtains the same environment (files, default directory, privileges, etc.) regardless of the node to which he or she is logged in. In many respects the VAXcluster feels like a single system to the user.

Figure 5 shows an example of a small VAXcluster and some of the major software components. At the highest level, multiple user processes on each node

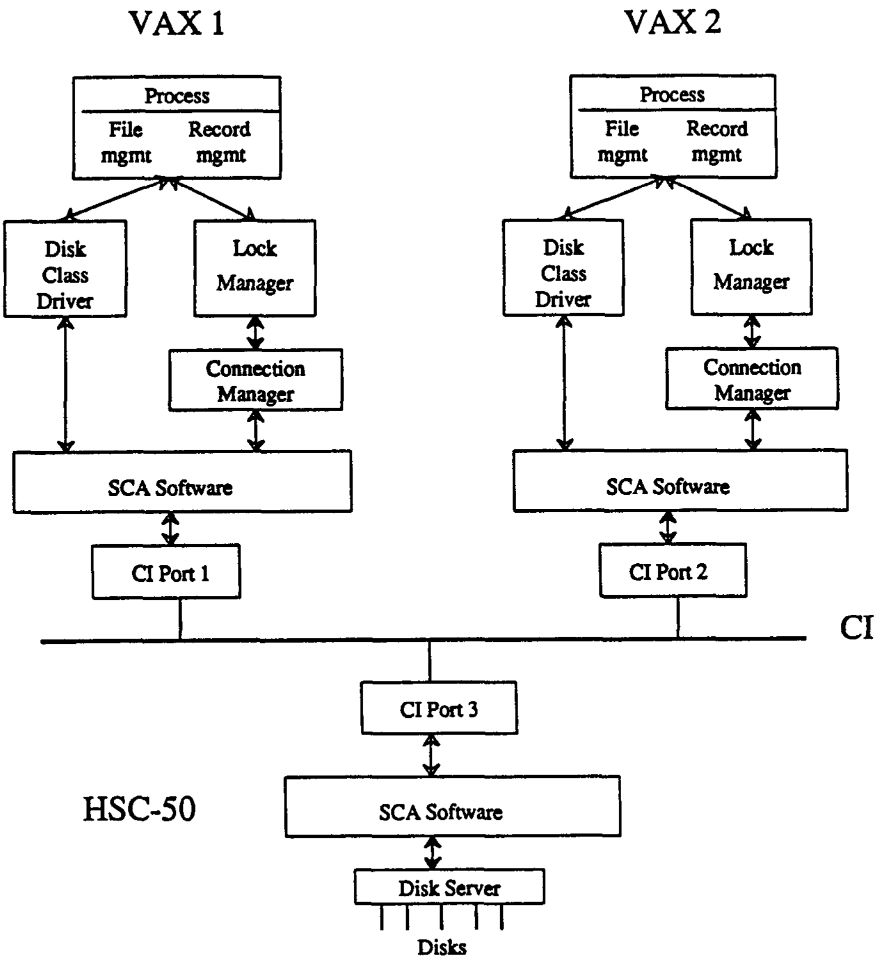


Fig. 5. VAXcluster software structure.

execute in separate address spaces. File and record management services are implemented as procedure-based code within each process.

The file and record services rely on lower level primitives such as the *lock manager* and *disk class driver*. The lock manager is the foundation of all resource sharing in both clustered and single-node VMS systems. It provides services for naming, locking, and unlocking cluster-wide resources. The disk class driver, mentioned earlier, uses MSCP to communicate with disk servers. The disk class driver runs in both clustered and nonclustered environments and contains no knowledge of the VAXcluster. SCA software below the driver is responsible for routing driver messages to the correct device controller.

Cluster *connection managers* are responsible for coordinating the cluster. Connection managers on all cluster nodes collectively decide upon cluster membership, which varies as nodes leave and join the cluster. Connection managers recognize recoverable failures in remote nodes; they provide data transfer services that handle such failures transparently to higher software levels.

5.1 Forming a Cluster

A VAXcluster is formed when a sufficient set of VAX nodes and mass storage resources become available. New nodes may boot and join the cluster, and members may fail or shut down and leave the cluster. When a node leaves or joins, the process of reforming the cluster is called a *cluster transition.* Cluster transitions are managed by the connection managers.

In an operating cluster, each connection manager has a list of all member nodes. The list must be agreed upon by all members. A single node can be a member of only one VAXcluster; in particular, the same resource (such as a disk controller) cannot be shared by two clusters or the integrity of the resources could not be guaranteed. Therefore, connection managers must prevent the partitioning of a cluster into two or more clusters attempting to share the same resources.

To prevent partitioning, VMS uses a quorum voting scheme. Each cluster node contributes a number of votes and the connection managers dynamically compute the total votes of all members. The connection managers also maintain a dynamic quorum value. As transitions occur, the cluster continues to run as long as the total votes present equals or exceeds the quorum. Should the total votes fall below the quorum, the connection managers suspend process activity throughout the cluster. When a node joins and brings the total votes up to the quorum, processes continue.

Initial vote and quorum values are set for each node by a system manager, and can be used to determine the smallest set of resources needed to operate as a VAXcluster. In order to start the cluster, each node contains an initial estimate of what the quorum should be. However, as nodes join, connection managers increase the quorum value, if necessary, so it is at least $(V + 2)/2$, where V is the current vote total.

A cluster member may have a recoverable error in its communications, that is, one that leaves the node's memory intact and allows the operating system to continue running after the error condition has disappeared. Such errors can cause termination of a virtual circuit and a corresponding loss in communication. When cluster members detect the loss of communication with a node, they wait for a short period for the failing member to reestablish contact. The waiting period is called the *reconnect interval* and the system manager sets it, usually to about a minute. If the failing member recovers within the reconnect interval, it rejoins the cluster. During the time the failing member is recovering, some processes experience a delay in service. If the failing member does not rejoin within the reconnect interval, surviving members remove it from the cluster and continue if sufficient votes are present. A node that recovers after it has been removed from the cluster is told to reboot by other connection managers.

5.2 Shared Files

The VAXcluster provides a cluster-wide shared file system to its users. Cluster accessible files can exist on CI-based disk controllers or on disks local to any of the cluster nodes (e.g., connected via Unibus or Massbus). Each cluster disk has a unique name. A complete cluster file name includes the disk device name, the directory name, and the file name. Using the device name for a file, cluster software can locate the node (either CPU or disk controller) on which the file resides.

Cluster file activity requires synchronization: exclusive-write file opens, coordination of file system data structures, and management of file system caches are a few examples. However, despite the fact that files can be shared cluster wide, file management services are largely *unaware* of whether they are executing in a clustered environment. The file managers synchronize through the VMS lock manager, to be described below. It is the lock manager that handles locking and unlocking of resources across the cluster. At the level of the file manager, then, cluster file sharing is similar to single-node file sharing. Lower levels handle cluster-wide synchronization and routing of physical-level disk requests to the correct device.

5.3 Distributed Lock Manager

As previously described, the VMS lock manager is the basis for cluster-wide synchronization. Several goals influenced the design of the lock manager for a distributed environment. First, programs using the lock manager must run in both single node and cluster configurations. Second, lock services must be efficient to support system-level software that makes frequent requests. Therefore, in a VAXcluster, the lock manager must minimize the number of SCA messages needed to manage locks. In a single node configuration, the lock manager must recognize the simpler environment and bypass cluster-specific overhead. Finally, the lock manager must recover from failures of nodes holding locks so that surviving nodes can continue to access shared resources in a consistent manner.

The VMS lock manager services allow cooperating processes to define shared resources and synchronize access to those resources. A resource can be any object an application cares to define. Each resource has a user-defined name by which it is referenced. The lock manager provides basic synchronization services to request a lock and release a lock. Each lock request specifies a locking mode, such as exclusive access, protected read, concurrent read, concurrent write, null, etc. If a process requests a lock that is incompatible with existing locks, the request is queued until the resource becomes available.

The lock manager provides a *lock conversion* service to change the locking mode on an existing lock. Conversions are faster than new lock requests because the database representing the lock already exists. For this reason, applications frequently hold a null lock (a no access place holder) on a resource and then convert it to a more restricted access mode later.

In many applications resources may be subdivided into a resource tree, as illustrated in Figure 6. In this example, the resource Disk Volume contains resources File 1 through File 3; resource File 3 contains resources Record 1 and Record 2, and so on. The first locking request for a resource can specify the parent of the resource, thereby defining its relationship in a tree. A process making several global changes can hold a high-level lock (e.g., the root) and make them all very efficiently; a process making a small low-level change (e.g., a leaf) can do so while still permitting concurrent access to other parts of the tree [8].

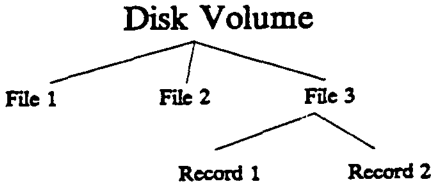


Fig. 6. VAXcluster locking structure.

The lock manager's implementation is intended to distribute the overhead of lock management throughout the cluster, while still minimizing the inter-node traffic needed to perform lock services. The database is therefore divided into two parts—the resource lock descriptions and the resource lock directory system—both of which are distributed. Each resource has a *master node* responsible for granting locks on the resource; the master maintains a list of granted locks and a queue of waiting requests for that resource. The master for all operations for a single tree is the node on which the lock request for the root was made. While the master maintains the lock data for its resource tree, any node holding a lock on a resource mastered by another node keeps its own copy of the resource and lock descriptions.

The second part of the database, the resource directory system, maps a resource name into the name of the master node for that resource. The directory database is distributed among nodes willing to share this overhead. Given a resource name, a node can trivially compute the responsible directory as a function of the name string and the number of directory nodes.

In order to lock a resource in a VAXcluster, the lock manager sends a lock request message via SCA to the directory for the resource. The directory responds in one of three ways:

(1) If the directory is located on the master node for the resource, it performs the lock request and sends a confirmation response to the requesting system.
(2) If the directory is not on the master node but finds the resource defined, it returns a response containing the identity of the master node.
(3) If the directory finds the resource to be undefined, it returns a response telling the requesting node to master the resource itself.

In the best cases (1 and 3) two messages are required to request a lock, whereas case 2 takes four messages. An unlock is executed with one message. If the lock request is for a subresource in a resource tree, the requesting process will either be located on the master node (i.e., the request is local) or will know who the master for its parent is, allowing it to bypass the directory lookup. In all cases, the number of messages required is independent of the number of nodes in the VAXcluster.

In addition to standard locking services, the lock manager supports data caching in a distributed environment. Depending on the frequency of modifications, caching of shared data in a distributed system can substantially reduce the I/O and communications workload.

A 16-byte block of information, called a *value block*, can be associated with a resource when the resource is defined to the lock manager. The value in the value block can be modified by a process releasing a lock on the resource, and read by a process when it acquires ownership. Thus, this information can be passed along with the resource ownership.

As an example, assume two processes are sharing some disk data and use the value block as a version number of the data. Each process caches the data in a local buffer and holds the version number of the cached data. To modify the data, a process locks it and compares the latest version returned with the lock with the version of the cached data. If they agree, the cache is valid; if the versions disagree, then the cache must be reloaded from disk. After modifying the data, the process writes the modified data back to disk and releases the lock. Releasing the lock increments the version number.

Another mechanism useful for caching data with deferred writeback is a software interrupt option. This is similar to the Mesa file system call-back mechanism [14], but is used for a different purpose. When requesting an exclusive lock, a process can specify that it should be notified by software interrupt if another lock request on the resource is forced to block. Continuing the example of cached disk data, a process owning the lock can cache and repeatedly modify the data. It writes the data back to disk and releases the lock when notified that it is blocking another process.

In the case of cluster transitions (e.g., failure of a node), the connection manager notifies the lock manager that a transition has started. Each lock manager performs recovery action, and all lock managers must complete this activity before cluster operation can continue.

As the first step in handling transitions, a lock manager deallocates all locks acquired on behalf of other systems. Only local lock and resource information is retained. Temporarily, there are no resource masters or directory nodes. In the second step, each lock manager reacquires each lock it had when the cluster transition began. This establishes new directory nodes based on a new set of eligible cluster members and rearranges the assignment of master nodes. If a node has left the cluster, the net result is to release locks held by that node. If no node has left the cluster but nodes have joined, this recovery is not necessary from an integrity point of view. However, it is performed to distribute directory and lock mastering overhead more fairly.

Some resources, depending on how they are modified, can be left in an inconsistent state by a cluster transition. One method of handling this problem would be to mark the locks for such resources to prevent access following a transition. A special process can then search for such locks and perform needed consistency checks before releasing them.

5.4 Batch And Print Services

In a VAXcluster, users may either submit a batch job to a queue on a particular node (not necessarily their own node), or submit a job to a cluster-wide batch queue. Jobs on the cluster-wide queue are routed to queues attached to specific nodes for execution. The algorithm for assigning jobs to specific nodes is a simple one based on the ratio of executing jobs compared to the job limit of the queue.

Management of batch jobs is the responsibility of a VMS process called the job controller. Each VMS node runs a job controller process. The process acquires work from one or more batch queues. Batch queues are stored in a disk file that may be shared by all nodes. Synchronization of queue manipulation is handled with lock manager services.

Print queues are similar to batch queues. Users may queue a request for a specific printer (not necessarily physically attached to their own node) or may let the operating system choose an available printer from those in the cluster.

Both batch and print jobs can be declared restartable. If a node fails, restartable jobs are either requeued to complete on another node in the cluster, or will execute when the failed node reboots (for jobs that required to execute on a specific node).

6. TERMINAL SUPPORT

The optimum method for connecting users' terminals to a VAXcluster is through the Local Area Terminal Server (LAT). Terminals are connected to LAT, which is attached to VAXs by Ethernet. Users command LAT to connect them to a specific node or to any node in the cluster. The ease of switching nodes leads users to find and use the least busy node. It also allows users to quickly move from a failed node to one that is still running. If LAT is directed to select a node, it attempts to find the least busy node. Its choice is based on node CPU type (a measure of processing power) and recent idle time.

7. PERFORMANCE

Performance of the system communications architecture (SCA) and the underlying hardware interconnect was measured using operating system processes running on clustered VAX nodes. By operating system processes, we mean that these programs run at the same level as the connection manager shown in Figure 5. These processes measured the datagram, message, and block transfer throughput for various message sizes. All numbers are approximate, as performance may vary due to different command options.

Table I shows the CPU time used on a VAX 11/780 to send and receive a reply to a 576-byte datagram and a 96-byte message. These sizes were used because they were representative of the sizes used by VMS. The CPU time to initiate a block transfer and receive the response is the same as that for a sequenced message (i.e., 320 μs) independent of the transfer size. Round trip elapsed time is also shown. In general, for a given transfer size the performance of datagrams and messages is approximately the same. The results also depend on the amount of buffering and the number of messages queued to the port before waiting for a response.

Table II shows the number of message round trips per second achieved on a cluster with between 2 and 12 communicating nodes. Each node was communicating with one other node; for example, in the 12 node case there were 6 pairs of communicating nodes, each pair engaged in sending and receiving a single stream of 112-byte messages. These experiments were run on VAX 11/785 nodes. As can be seen from Table II, the total number of messages increases nearly linearly. Once again, performance depends on the number of messages queued at a time. While one message was queued at a time in this experiment, previous experiments have shown that by queueing four messages at a time, it is possible to achieve 3000 message round trips per second on two VAX 11/780s.

Table I

	Datagram	Message
Send/receive	290 μs	320 μs
CPU time	(576 byte)	(96 byte)
Send/receive	1500 μs	1000 μs
Round trip	(576 byte)	(96 byte)
Elapsed time		

Table II

Number of nodes	Cluster round-trip (messages/s)
2	1090
4	2170
6	3250
8	4330
10	5420
12	6480

Table III

Number of nodes	Cluster block transfer thruput (mbytes/s)
2	2.65
4	5.28
6	7.70

Table IV

	Single-CPU system	Clustered system
Request lock	400 μs	3000 μs
Convert lock	250 μs	1000–2300 μs
Release lock	300 μs	700 μs

Throughput results are shown in Table III. In this experiment, 4 streams of 64-kbyte block transfers were initiated between 2, 4, and 6 VAX 11/785 nodes. Again, throughput increases almost linearly. Throughput measurements for datagrams and messages show that a throughput of about 2.4 Mbytes per second can be achieved between two VAX 11/780 nodes sending 900-byte messages or datagrams.

These throughputs do not represent real workloads, which produce less traffic than the contrived test programs. It is clear that the 8.75 Mbyte per second (per path) CI is not the bottleneck. For this test, we believe that the limiting factor is the speed of the CI port and the memory bandwidth of the host. Of course, the reason for having a high-speed interconnect is to provide bandwidth for multiple hosts.

This performance provides a basis for efficient execution of higher level distributed services, such as the lock manager. Table IV lists approximate performance values for lock operations in both clustered and nonclustered environments. As previously stated, lock conversions are faster than new lock requests because the database already exists. The measured elapsed time for a conversion depends upon the existing mode and new mode.

8. CONCLUSIONS

A principal goal of VAXclusters was the development of an available and extensible multicomputer system built from standard processors and a general-purpose operating system. Much was gained by the joint design of distributed software, communications protocols, and hardware aimed to meet this goal. For example:

(1) The CI interconnect supports fast message transfer needed by system software.
(2) The CI port implements many of the functions needed by SCA software.
(3) The HSC-50 with its message protcol and request queueing optimization logic provides direct network access to a large pool of disks for multiple hosts.

Designing hardware and software together allows for system-level tradeoffs; the software interface and protocols can be tuned to the hardware devices.

An important simplifying aspect of the VAXcluster design is the use of a distributed lock manager for resource synchronization. In this way, higher level services such as the file system do not require special code to handle sharing in a distributed environment. However, performance of the lock manager becomes a crucial factor. Distributed lock manager performance has been attacked with the design of a locking protocol requiring a fixed number of messages, independent of the number of cooperating nodes.

Finally, we believe that the performance measurements presented show the extent to which the VAXcluster system has succeeded in implementing an efficient communications architecture. These numbers are particularly impressive when considering that VMS is a large, general-purpose operating system.

ACKNOWLEDGMENTS

VAXclusters are the result of work done by many individuals in several engineering groups at Digital Equipment Corporation. We would particularly like to acknowledge the contributions of Richard I. Hustvedt.

We wish to thank Bill Laing, Steve Neupauer, and Stan Amway for performance measurements reported in this paper, and Steve Beckhardt, Liz Hunt, Jim Krycka, Christian Saether, and Dave Thiel for reviews of earlier versions of the paper.

REFERENCES

1. ALMES, G. T., BLACK, A. P., LAZOWSKA, E. D., NOE, J. D. The eden system: A technical review. *IEEE Trans. Softw. Eng. SE-11*, 1 (Jan. 1985), 43–59.
2. APOLLO COMPUTER CORP. *Apollo domain architecture.* Tech. Rep. Apollo Computer Corporation, North Billerica, Mass., 1981.
3. BARTLETT, J. F. A nonstop kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16), ACM, New York, 1981, pp. 22–29.

4. BORG, A., BAUMBACH, J., AND GLAZER, S. A message system supporting fault tolerance. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct. 11–13), ACM, New York, 1983, pp. 90–99.
5. BROWNBIRDGE, A., MARSHALL, A., AND RANDELL, A. The newcastle connection or unixes of the world unite! *Softw.—Pract. Exp. 12* (1982), 1147–1162.
6. CHERITON, D., AND ZWAENEPOEL, W. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct. 11–13), ACM, New York, 1983, pp. 129–140.
7. FIELLAND, G., AND RODGERS, D. 32-bit computer system shares load equally among up to 12 processors. *Elect. Des.* (Sept. 1984), 153–168.
8. GRAY, J. N., LORIE, R. A., PUTZOLU, G. R., AND TRAIGER, I. L. Granularity of locks and degrees of consistency in a shared data base. In *Modelling in Data Base Management Systems.* Nijssen, G. M., Ed., North Holland, Amsterdam, 1976.
9. HWANG, K., AND BRIGGS, F. A. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.
10. KATSUKI, D., ELSAM, E. S., MANN, W. F., ROBERTS, E. S., ROBINSON, J. G., SKOWRONSKI, F. S., WOLF, E. W. Pluribus—An operational fault-tolerant multiprocessor. In *Proceedings of the IEEE 66*, 10 (Oct. 1978), 1146–1159.
11. KATZMAN, J. A. The Tandem 16: A fault-tolerant computing system. In *Computer Structures: Principles and Examples*, Siewiorek, D. P., Bell, C. G., and Newell, A., Eds., McGraw-Hill, New York, 1982, pp. 470–480.
12. METCALFE, R. M., AND BOGGS, D. R. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM 19*, 7 (July 1976), 395–404.
13. POPEK, G., WALKER, B., CHOW, J., EDWARDS, D., KLINE, C., RUDISIN, G., THIEL, G. LOCUS: A network transparent, high reliability distributed system. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16), ACM, New York, 1981, pp. 169–177.
14. REID, L. G., AND KARLTON, P. L. A file system supporting cooperation between programs. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct. 11–13), ACM, New York, 1983, pp. 20–29.
15. SATYANARAYANAN, M. *Multiprocessors: A Comparative Study.* Prentice-Hall, Englewood Cliffs, N.J., 1980.
16. STRECKER, W. D. VAX-11/780: A virtual address extension to the DEC PDP-11 family. In *Proceedings of AFIPS NCC*, 1978, pp. 967–980.

Received July 1985; revised December 1985; accepted December 1985